Software Certification Management How Can Formal Methods Help?

Dieter Hutter

German Research Center for Artificial Intelligence (DFKI GmbH)
Saarbrücken, Germany

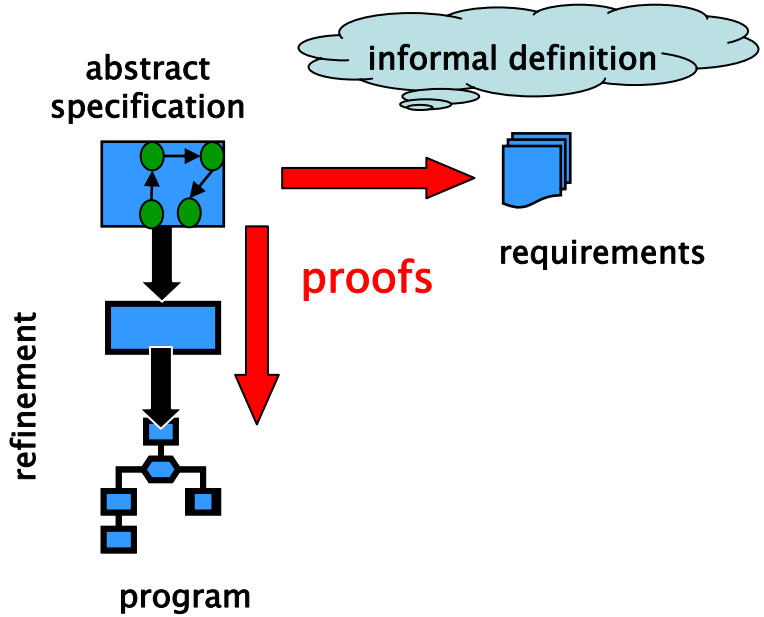
Software Certification Management

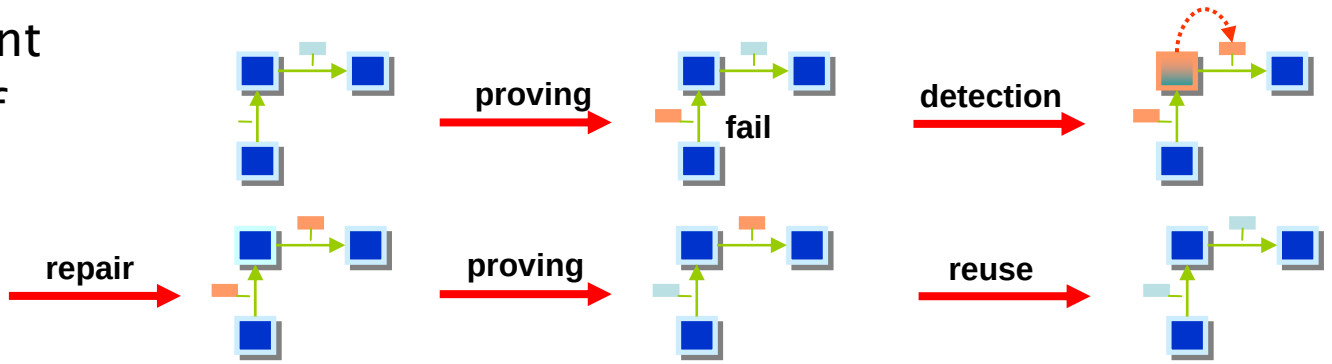
Management of dependency and consistency

Static dependencies:
different layers of specifications
- formal verification

Dynamic dependencies:
changing parts of the development
- management of change

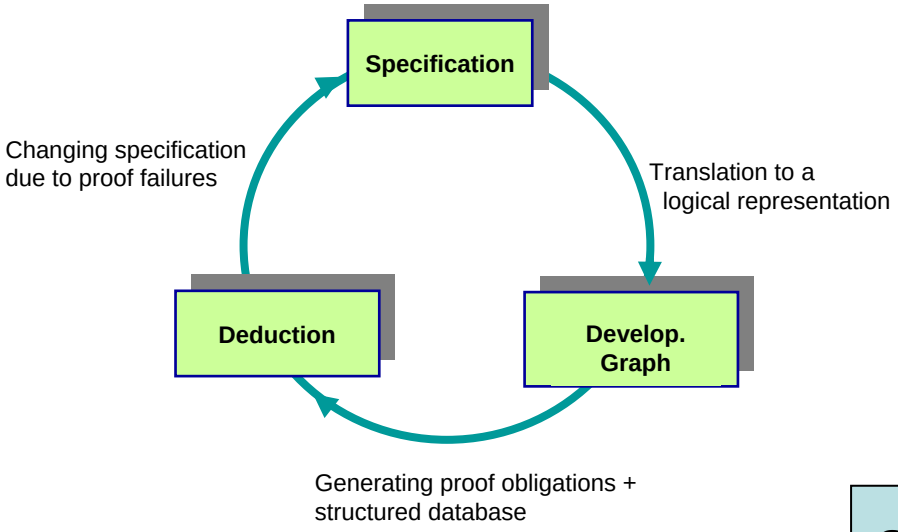
distributed development
- Merge/Patch/Diff

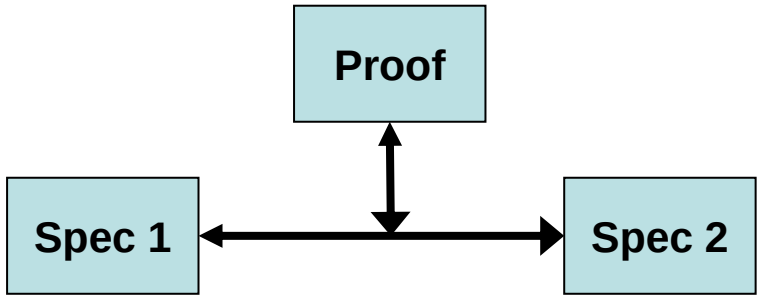




Dynamic Dependencies

Formal management of change



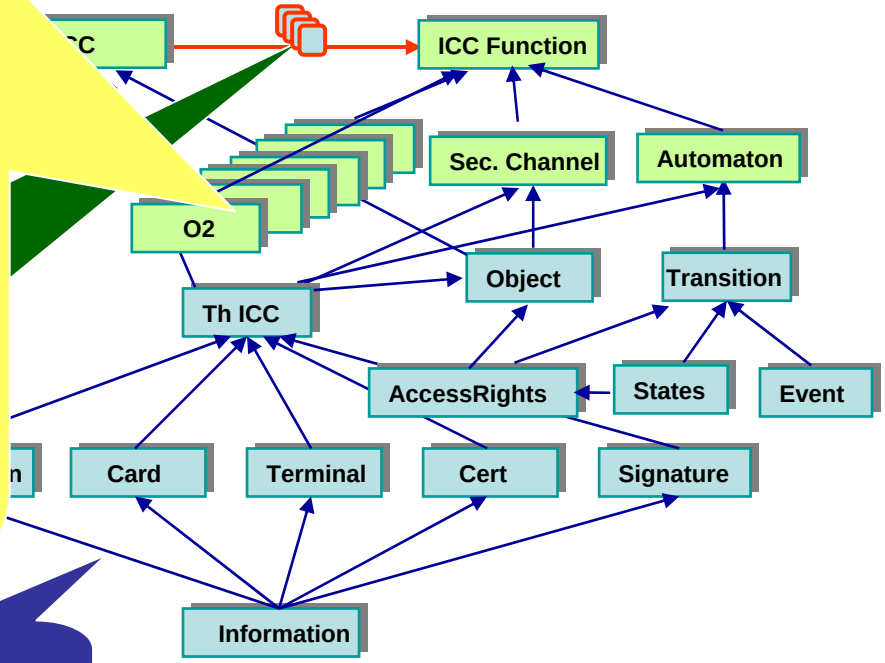


„Redundancy“ by formal proofs

Formal Developments as Structured Objects

Axioms, Logic, Calculus

```
o5.buck = tbucks.bckobjects and o5.buck' = tbucks.bckautomaton and
  (EX  o5.newvalue : (o5.command = tterminal.ifddomodify(tobjectids.obj5,
                                                         o5.newvalue)

taccessrights.allowed(tobjectids.obj5, o5.state, taccessrights.armodify)
            -> (o5.value‘ = tmaybe{tinformation.information}.def (o5.newvalue) and
                o5.valueout' = tcard.answermodified))

( not taccessrights.allowed(tobjectids.obj5, o5.state,  taccessrights.armodify)
     -> ( o5.value = o5.value and o5.valueout' = tcard.answerdenied)))

o5.buck = tbucks.bckobjects and o5.buck' = tbucks.bckautomaton
        and o5.command = tterminal.ifddoread(tobjectids.obj5)
        and (  taccessrights.allowed(tobjectids.obj5, o5.state,  taccessrights.arread)
    -> o5.valueout' = o5.value)
         and ( not taccessrights.allowed(tobjectids.obj5, o5.state,  taccessrights.arread)
                -> o5.valueout' = tcard.answerdenied)
        and o5.value = o5.value'))_{(o5.value, o5.valueout, o5.buck)}

o5.buck = tbucks.bckobjects and o5.buck' = tbucks.bckautomaton
 and (EX  o5.i, o5.j  :
           (  o5.command = tterminal.ifddoverify(o5.i, o5.j)
             and (   taccessrights.allowed(tobjectids.obj5, o5.state, taccessrights.aruse)
                  ->  ( o5.valueout' = tcard.answersuccess
                        or o5.valueout' =  tcard.answerfailure ))  ...
```



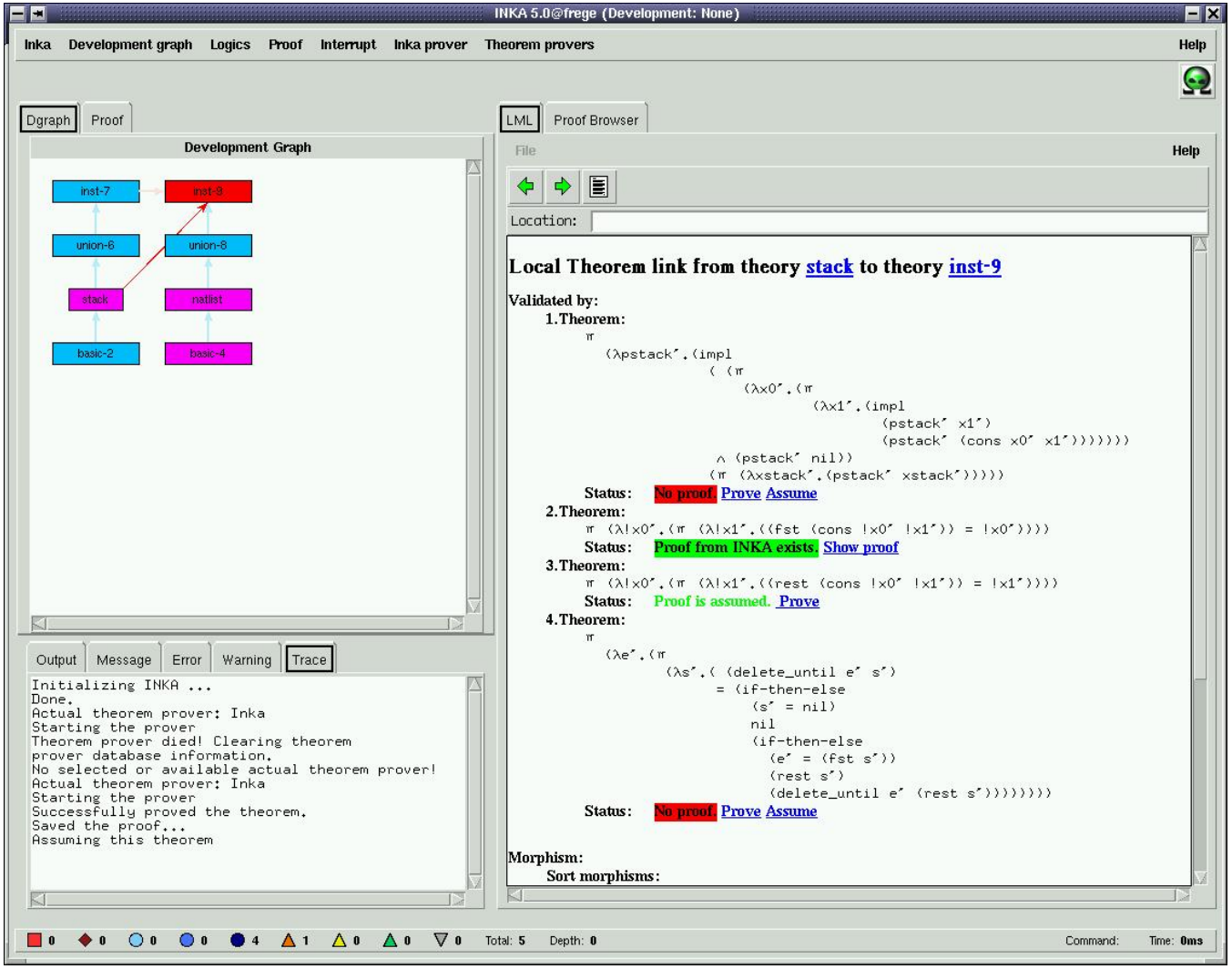
Signature morphisms

Verification of Properties

- Types of „properties“:
  - Structured properties: decomposition
  - Elementary properties:
    formal or „informal“ proof

- Decomposition und composition:
  - Properties are decomposed according to the structure of the doucments
  - Reuse of properties of unchanged objects
  - Synthesis of properties for changed or new objects

MAYA – Managing Formal Developments



MAYA - Specification

emacs: /usr/X11/bin/xemacs [21.1 (patch 14) "Cuyahoga Valley" XEmacs Lucid] AMAST.casl

File Edit Mule Apps Options Buffers Tools Top <<< . >>> Bot Help

Open Dired Save Print Cut Copy Paste Undo Spell Replace Mail Info Compile Debug News

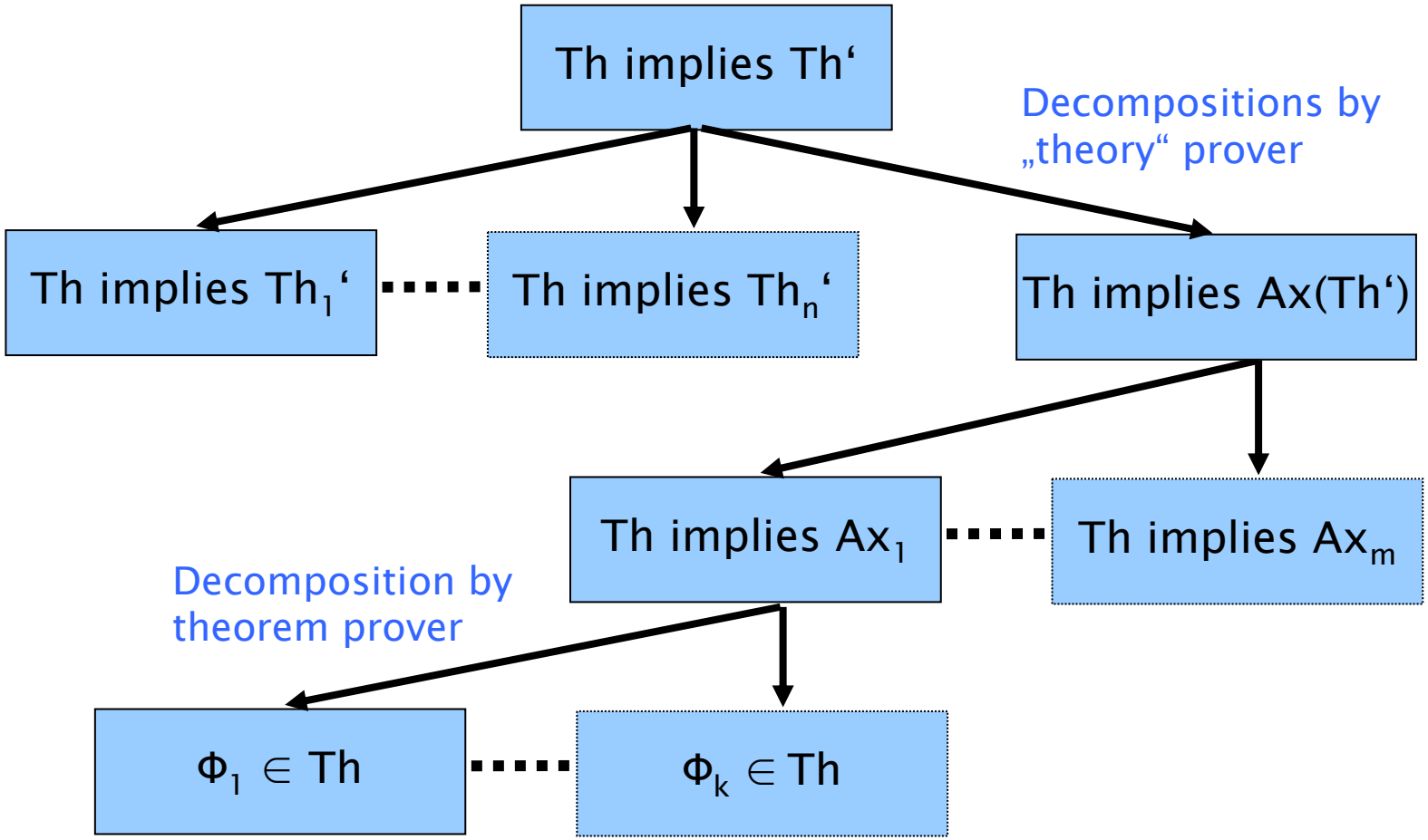
```
spec natlist =
   {
    generated type nat ::= null | s(p:nat);
    var x,y,z:nat;
    op * : nat * nat -> nat, comm, assoc, unit s(null);
    op +(x:nat; y:nat):nat =
       y when x = null
       else s(+(p(x), y));

    axiom +(x,y) = +(y,x);
    axiom +(x,+(y,z)) = +(+(x,y),z);
   }

  then

  {
   generated type natlist ::= nil
                            | cons(fst:nat; rest:natlist);

   var l1,l2:natlist;
   var n1,n2:nat;

   op app : natlist * natlist -> natlist, assoc, unit nil;

   axiom app(cons(n1,l1),l2) = cons(n1, app(l1,l2));

   op addlast(n:nat; l:natlist):natlist =
      cons(n,nil) when l = nil
      else cons(fst(l), addlast(n,rest(l)));

   op delete_until(n:nat; l:natlist):natlist =
      nil when l = nil
      else rest(l) when n = fst(l)
           else delete_until(n, rest(l));
}
```

Noconv-----XEmacs: AMAST.casl (Fundamental PenDel)----To

```
spec stack =
   {
    sort elem;
   }

then

   {
    generated type stack ::= empty_stack
                           | push(top:elem; pop:stack);

    op poprec(e:elem; s:stack):stack =
       empty_stack when s = empty_stack
       else pop(s) when e = top(s)
            else poprec(e, pop(s));
   }

view viewit : stack to natlist =
     sorts elem |-> nat,
           stack |-> natlist,
     ops   poprec:elem * stack ->stack  |-> delete_until,
           empty_stack:stack            |-> nil,
           top: stack -> elem           |-> fst,
           pop: stack -> stack          |-> rest,
           push:elem * stack -> stack   |-> cons
end
```

Noconv-----XEmacs: AMAST.casl (Fundamental PenDel)----

Structural Decomposition of Proof Obligations

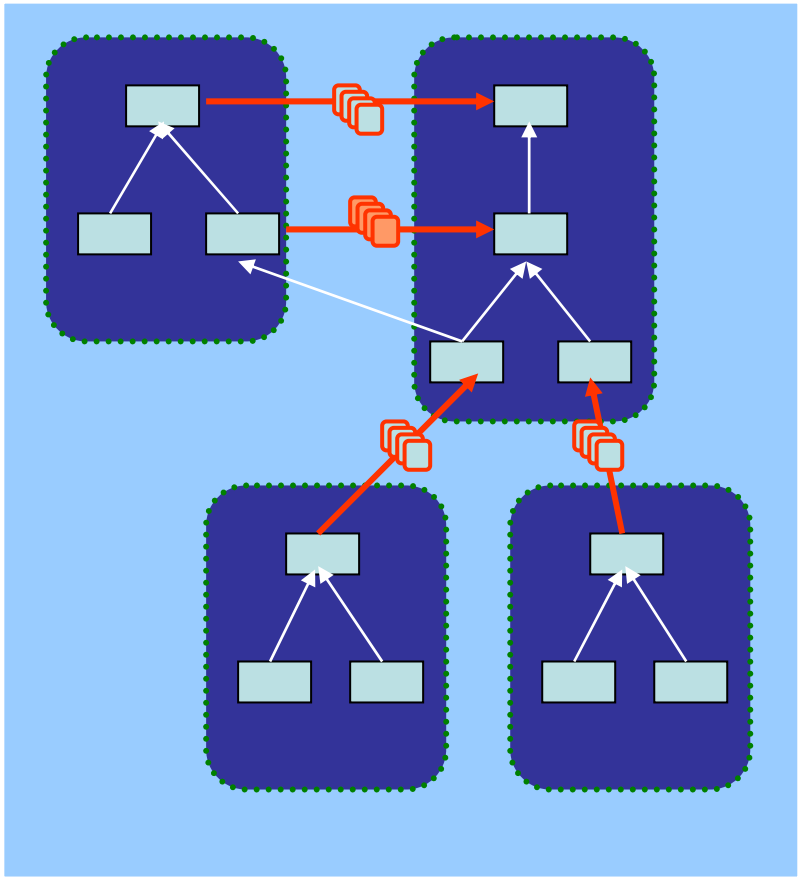


Example: Development Graphs

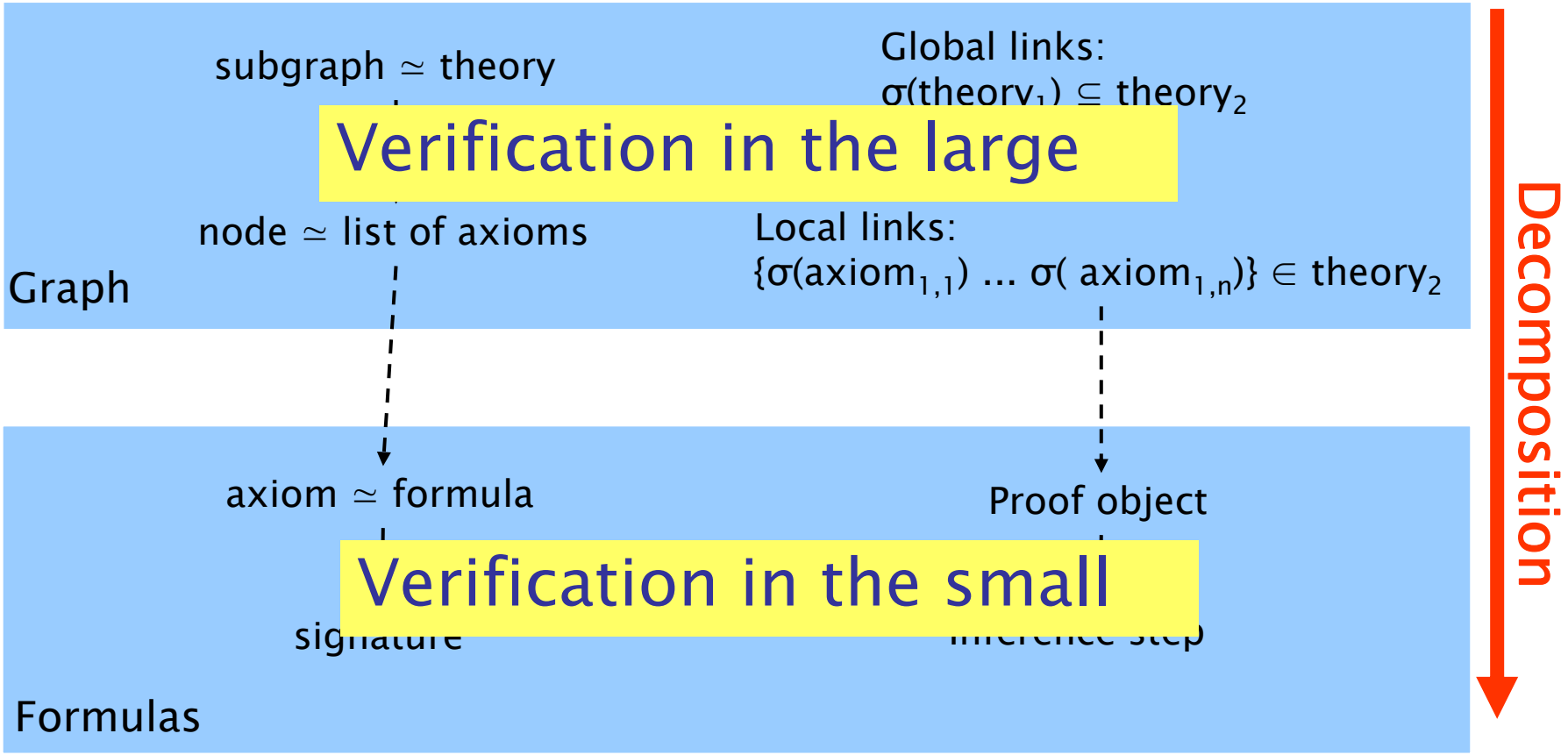
- Logic based representation of structured formal developments
- Specifications and implementations as theories (consequence relations)
- Formal relations between parts of developments (morphisms)
- supports different formalisms (logics) to represent different parts

Now used

- to define proof theory of CASL
- to specify structuring in OMDoc



Structuring Mechanisms in Formal Methods



Lessons Learned

- Structured objects:
  - E.g. theories, formulas, terms, signature
  - E.g. document, chapter, section, paragraph
  - Acyclic graphs as object representation

- Structured properties between objects:
  - E.g. satisfies_{Th}, $\text{satisfies}_{Ax(Th)}$, satisfies_{ϕ}

- Decomposition rules along object structure
  - E.g. satisfies_{Th} by using $\text{satisfies}_{Ax(Th)}$ for all subtheories

- Calculi to prove properties on various levels

- Rules to adapt inference steps in case of changes

Distibuted Development

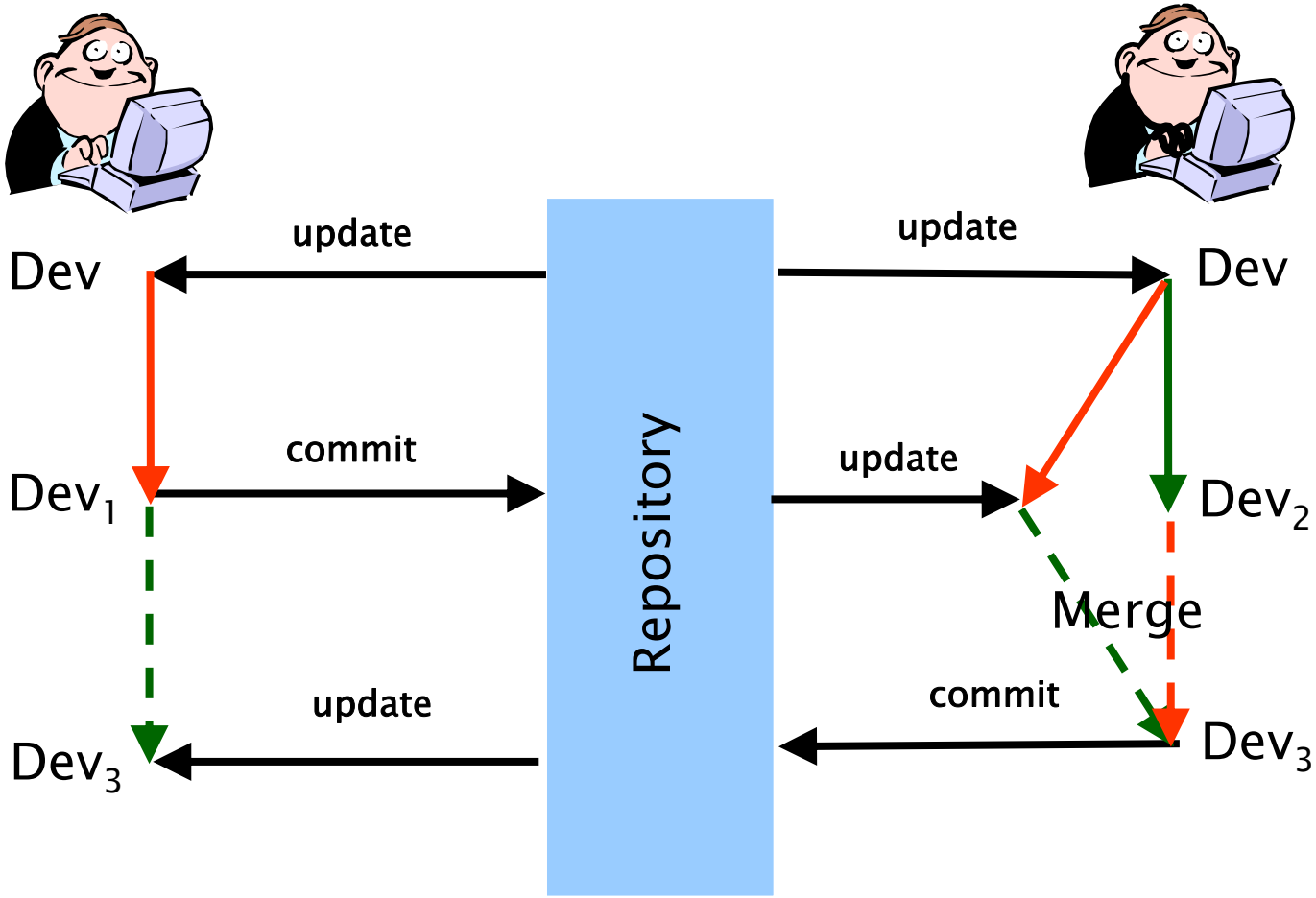
- Distributed development
  - Update of local developments
  - Merge of different branches
  - Notion of conflicting developments

  ⇒ Integration of different specifications

- Analysis, retrieval and repair of derived properties
  - Reuse of proofs
  - Transfer of informal knowledge

  ⇒Translation of proof work in common development

Distributed Development (CVS)



Consistency of (Distributed) Developments

- Development as a collection of various (types of) documents

- By „consistency“ we mean
  - Preserving syntactical correctness
  - Preserving the static semantics
  - Preserving proofs (properties)
    e.g.:
    - Implementation satisfies requirement specification
    - Specification ensures security requirements
    - Dependencies in the documentation of the project

Merging Distributed Developments

- CVS: conflict occurs iff the same text-line is changed in both developments

- Using structured objects:
  - Non-local effects of changes!
  - General rule:

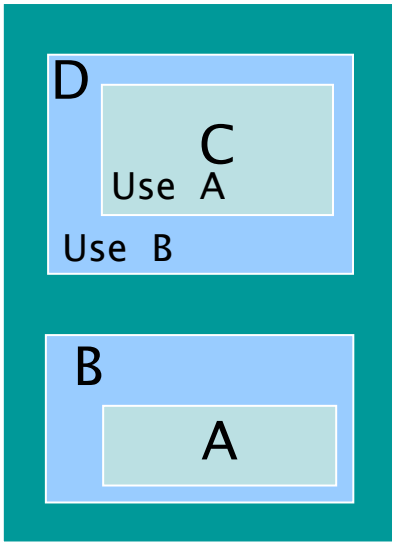
- Single-worker rule:
  conflict occurs if a developer inserts or edits an object that depends on a object changed or deleted by another developer

Decomposition of Semantic Conflicts

- Containment defines structuring of objects

- Decomposition rules to unfold dependency of composed objects into dependencies of subobjects:

  e.g. B < D into (A < C, ...)



- Instead demanding single-worker-rule for B < D we demand single-worker-rule for A < C, ...

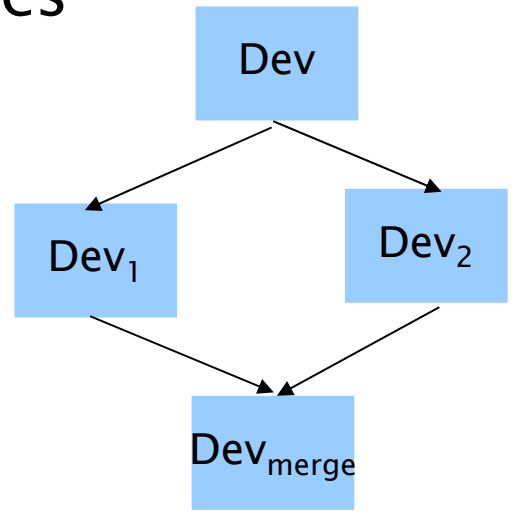
What is a Semantic Conflict ?

conflict occurs if a developer inserts or edits an object that depends on a object changed or deleted by another developer

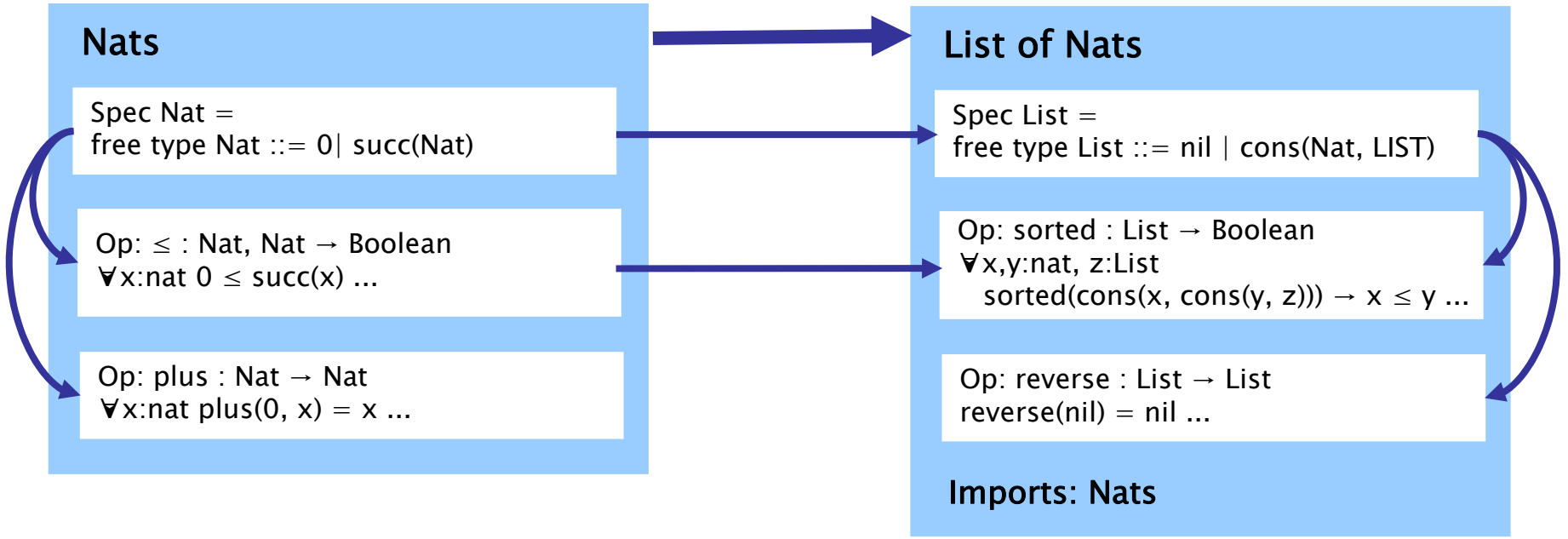
⇒ no randomly generated dependencies are allowed (single-worker-rule):

$B_{merge} < D_{merge}$ implies

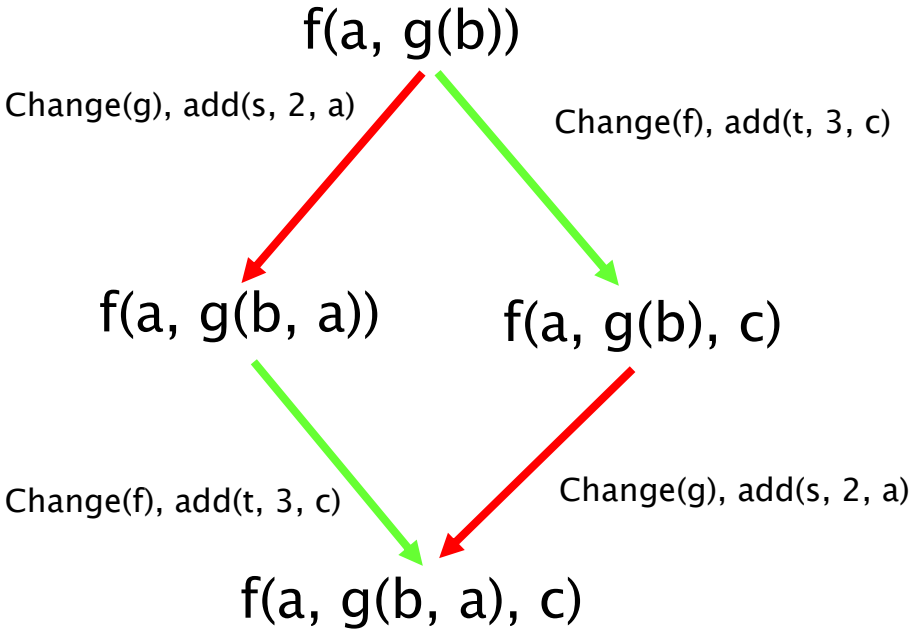
$$(B_{merge} = B_1 \wedge D_{merge} = D_1\ ) \vee$$
$$(B_{merge} = B_2 \wedge D_{merge} = D_2\ )$$

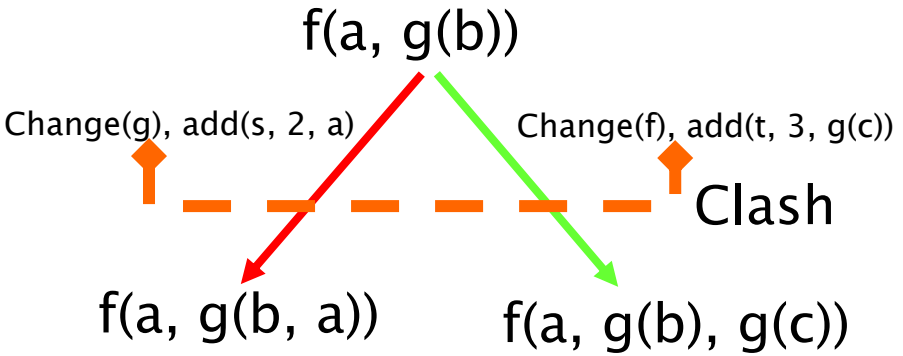


Dependencies in MAYA



Decomposition in its Extreme





Semantic clash: no merge possible!

Conclusion

- Formal methods can help !!!
- Helps for a formal semantics for decomposing and composing certifications
- Formal semantics for individual „certificates“
- Support for a management of change
  - proofs as formal representation of certificates
  - effects of changes